Computer Science • 14 (1) 2013 http://dx.doi.org/10.7494/csci.2013.14.1.129

Małgorzata Wielgus
Przemysław Dąbek
Roman Janusz
Tomasz Kowal
Wojciech Turek

ERLANG-BASED SOFTWARE UPDATE PLATFORM FOR MOBILE DEVICES

Abstract *Growing computational power of mobile devices modifies existing approaches to data processing in large-scale sensor networks. Since sensors are no longer limited to simple data acquisition tasks, such networks can be considered complex geo-distributed data processing systems. Features and requirements of such systems justify use of Erlang language and technology for programming mobile devices. The technology provides several crucial features, including fault-tolerance, message-passing concurrency or hot-code loading. In this paper the problem of software management in Erlang-based distributed systems is discussed. A mechanism for installing and upgrading Erlang applications using operating system package manager is described. A platform for updating software in large scale systems is presented.*

Keywords Erlang, software updates, distributed system

1. Introduction

Fast development of mobile computers creates new domain of applications for large systems composed of many geo-distributed, interconnected devices. Mobile sensors are no longer limited to simple tasks related to data acquisition. Relatively high computational power of embedded systems makes it possible to execute complex algorithms. A sensor can process collected data, analyze image, sound or other signal, detect important information and notify when particular situation occurs. This type of sensor can be called active or intelligent. It provides information, not data.

System composed of active sensors provides high computational power in locations, where data appears. This removes limitations created by throughput of network, and makes it possible to create large scale systems, which can perform complex processing of all collected data. There are many areas of applications for such systems, for example:

Finding particular people or cars. A mobile devices equipped with cameras can execute image recognition algorithms. Configuration can define which plate numbers or faces are searched for. The device can report only, when suspected object is found.

Management of geographically spread devices. Devices can be installed in company cars for monitoring location and speed. A device can alarm when a car moves too fast or leaves specified area. Another example is monitoring of proper functioning of other devices, like GSM network base transceiver stations or oil pumpjacks.

Active security systems. Monitoring of large areas or buildings could be much more efficient if advanced processing of signals was performed continuously. Such system could allow particular people to enter specified zones or report unusual behaviors in particular areas.

Robotics. Embedded devices controlling robots are probably most advanced example of distributed computer systems.

This new possibilities require use of new architectures of large scale distributed systems composed of mobile devices. With computation moved to remote devices, amount and type of exchanged information significantly changes. Less data is passed from a device which reduces requirements concerning network throughput. However, flexibility of such solutions require creating methods for changing software parameters and updating algorithms on remote devices.

The problem of updating software in geo-distributed systems has been investigated by researchers working on sensor networks. Problems of efficient code propagation in sensor network has been discussed in [1] and [2]. Problems concerning security of updating software of sensors have been discussed in [3]. A survey of architectures for updating software in enterprise-scale networks can be found in [4].

Several proprietary system dedicated for management of particular brands of mobile devices have been created [5, 6]. These solutions are designed to manage software

on mobile phones or tablets used by employees of a particular company. This kind of devices and systems could be used for performing some of aforementioned tasks, however flexibility of such solution is limited.

Execution of complex applications can always result in high level of unexpected errors. In considered class of distributed systems, where large number of devices runs without manual supervision, high availability and error-recovery mechanisms should be supported. This requirement can be easily met using the Erlang technology [7], which is dedicated for creation high availability distributed systems. Other features of Erlang also seem suitable for creation of such systems, therefore the attempt of utilizing the technology seems justified.

Erlang-based large scale geo-distributed systems require tools and methods for efficient management, which is the main goal of the work described in this paper. In the next section main features of the Erlang technology will be described. Following sections will present assumptions, architecture and tests of the software update platform for Erlang-based distributed systems.

2. Erlang technology overview

Erlang is a technology and a programming language that mixes functional programming with an approach to easily build heavily parallel and distributed, highly available systems. It achieves these goals using a set of unique features, including:

- Virtual machine implementing message-passing concurrency model with lightweight Erlang processes.
- Built-in, language-integrated engine for communication in a distributed environment.
- Hot code swapping with a fine control over the software upgrade process. The aim of these is to allow an upgrade to be performed automatically without stopping any services.
- Fault-tolerance features. The most important one is supervisor behaviour for writing special control processes responsible for monitoring other processes and reacting accordingly when they crash. This allows programmers to take an *happy case programming* approach which means that they can ignore any exceptions as long as they don't have to be handled explicitly in some special way.
- Takeover and failover mechanisms for cluster systems.

2.1. Erlang OTP

Erlang OTP (Open Telecom Platform [8]) is an Erlang distribution released by Ericsson in 1998 when the language became open source. OTP is a set of standard Erlang libraries and corresponding, well-defined design principles for Erlang developers. OTP defines patterns for basic elements that make up the software as well as the general layout of completed, deployed environment.

Erlang is a technology that from the very beginning incorporates patterns, conceptions and approaches that are crucial for heavily parallel and distributed systems. Since such systems are becoming ubiquitous these days, Erlang definitely has the chance to become a technology of the future for large, distributed computer systems.

In this article, brief description of basic OTP principles will be presented with more attention on the ones important for the Software Update Platform.

OTP principles include:

- Supervisors and supervision trees
  Erlang software can be thought of as a set of lightweight Erlang processes communicating with each other. In supervision tree principle, these processes form a tree where the leaves are called workers and are doing the actual job, while other nodes are called supervisors. Each supervisor is responsible for monitoring its children and reacting accordingly when any of them crashes. Supervisors allow to design well-structured and fault-tolerant software.
- Behaviours
  Behaviours are a set of basic design patterns used to build common types of software pieces. Fundamental behaviours are:
  - `gen_server` for implementing simple servers and client-server relation between Erlang processes,
  - `gen_fsm` for implementing generic finite state machines,
  - `gen_event` for implementing event handling subsystems.
- Applications and releases
  These patterns define general layout of a self-contained, deployable piece of Erlang software. Applications and releases will be described in the next section as the Software Update Platform deals heavily with them.

2.2. Applications and Releases in Erlang

A deployable package of software written in Erlang is called an **embedded node**. An embedded node is a self-contained, configured Erlang environment along with actual software written in Erlang that can be deployed and run using a simple command. An embedded node contains:

- Erlang Runtime System (ERTS),
- a set of Erlang applications, the actual code,
- configuration for ERTS and applications,
- an Erlang release.

2.2.1. Erlang applications

An Erlang application is an independent piece of software that serves some particular functionality. An application is defined by its name, version, code (set of modules), dependencies (other applications) and other more finegrained settings and attributes. These are all configured in an `.app` file. Every application defines a way of starting

it, stopping it and possibly upgrading or downgrading it to another version (optional `appup` file). It also has its own piece of configuration. A running application is often made up of a single supervision tree.

2.2.2. Erlang release

An Erlang release is a configuration of what an embedded node contains and how it is started, stopped and upgraded. Thus, an Erlang release, defined by an `.rel` file, states which version of ERTS should be used in the node, lists a set of applications in particular versions that should be part of the release, and defines one or more way the Erlang node is started and stopped. Separate, optional `relup` file defines how the release is upgraded or downgraded (without stopping the node). Erlang release has its own version number.

2.3. Upgrading Erlang software

As a part of focus for high available systems, Erlang supports hot code swapping and very finegrained control over the upgrade process without stopping running node. Every application may define how its version should be changed to higher or lower in the `appup` file. Based on a set of `appup` files, a `relup` file may be generated which merges all operations listed in `appup` files into one big script that upgrades or downgrades the whole release. This script may be executed using standard Erlang API for release handling (the `release_handler` module).

Because each application is responsible for defining how its version should be changed, the upgrade process is very straightforward and requires only a few calls to the release handling functions. Thus, it can be easily performed without human interaction, by automatic tools.

Still, there are some problems with that method. Mainly, it is low-level as it requires calling the Erlang API on the target node. What is needed and what our platform aims to provide is a way of easy installation and deinstallation of the Erlang node on the target system using simple tools like package managers and also a way of easy management of a large number of devices.

3. Packaging of Erlang software

'Packaging of Erlang software' means a way of putting the contents of Erlang node into packages like `.deb`. These packages are easy to install, upgrade and remove with standard package manager commands, available on most operating systems.

The Software Update Platform described in this paper uses packages and a package manager as a primary mechanism for modifying software on particular devices. Creation and installation of such packages is not trivial, especially when it is very important to preserve ability to perform upgrades without stopping the Erlang node, using hot code swapping mechanism.

There are a few reasons why adopting existing packaging mechanism is justified:

- easy installation and deinstallation of Erlang node on the target system using a single command,
- reduction of amount of data downloaded during the upgrade (only the actually changed packages are fetched by the package manager),
- overall better integration with the target system,
- knowledge of Erlang technology is not required to perform installation or update.

3.1. Implementation details

The implementation aimed at creating tools to build Debian packages `.deb` that would span the contents of the Erlang node and create a set of packages ready to be pushed to a repository and easily installed on the target device. This also required general implementation of Debian maintainer scripts included into these packages. These scripts are responsible for management of the node during installation. Most importantly, they contain the code that performs the upgrade process.

3.1.1. Decomposition of the Erlang node into Debian package set

The Erlang node is decomposed into a set of Debian packages in the following way:

- Base package contains ERTS and its version is equal to the ERTS version. This package is architecture-dependent. It has no dependencies.
- Each Erlang application gets its own package. Its version is equal to the application version. Package dependencies reflect the list of dependent applications in the `.app` file.
- Erlang release files are packaged into the final, main package. This package is dependent on the base package and packages for all applications contained in the release. These dependencies are strict in terms of version of dependent packages – the release requires a concrete version of every application as well as the ERTS.

The only package that should be explicitly maintained by system administrator is the main package containing the release files. This package, thanks to its well-defined dependencies, represents the whole Erlang node and its installation will cause the whole node to be deployed on the device.

The point of splitting up the node into several packages related through dependencies was to reduce the amount of data downloaded during the upgrade process. It is a very common situation when only one Erlang application is to be upgraded. Standard Erlang features do not allow upgrading particular applications independently – a new version of the whole release must be created and upgraded. Without proper decomposition of the contents of the release, this would force downloading a big package containing every application, even though most of them did not change.

Usage of package manager solves this problem. When the node is decomposed into several packages, upgrade of the main package forces the package manager to download only these application packages whose version changed. This is all thanks to automatic dependency resolution provided by the package manager.

3.1.2. Upgrade of the release

There are a few maintainer scripts (see [9]) in all the packages spanning the Erlang node. They are responsible for starting the node when it gets installed, stopping it when it gets removed and, most importantly, performing the release upgrade process when the main package is being upgraded by the package manager.

The script performing the release upgrade checks whether the node is running, using tools available in the node `bin` directory. When the node is running, the script remotely calls appropriate Erlang code on the running node, causing it to switch to the higher version of the release (standard Erlang release upgrade using the `release_handler`, without stopping the node). Hot upgrade may fail or the node may have been down from the beginning. If so, a manual replacement of the old version of the release with the new version is performed (manual replacement of some files) and the node is started.

3.2. Results and effects

All crucial features regarding integration of package manager with Software Update Platform have been successfully implemented.

From the point of view of an Erlang developer maintaining the software being upgraded, the main feature implemented are several helpful scripts. They allow easy generation of Debian packages from an Erlang release generated with `rebar` (more information about development model can be found in 4.2). There is also a script for easy generation of the `relup` file.

From the point of view of a target system administrator, the main advantages of package manager are mentioned earlier. The most important is the easy way to install, upgrade and remove Erlang nodes. The only configuration needed at the target system is addition of Debian repository in the `apt` configuration file.

From the point of view of an user of the Software Update Platform, main feature implemented is a built-in Debian packages repository along with an UI to manage its contents.

We were also successful in obtaining desired non-functional features of the system regarding its use of package manager. This includes smaller downloads from the server to the target devices and ability to use hot-code swapping by the package manager during upgrade process.

3.3. Problems and limitations

Although the integration of package was successful, due to a significant mismatch between how package manager works and how standard Erlang upgrade tools work, some workarounds were required.

The main problem encountered regarded ability to perform hot-upgrade without stopping the system. Erlang upgrade tools require that at the point of a release upgrade, the whole old version of the release and the whole new version of the release

are present in the filesystem. When the release is split into several packages, it is impossible to find a moment where this requirement is met. It is also impossible to influence the way package manager works to force it to behave as we wanted.

Because of that, a workaround had to be introduced. Erlang application and Erlang release packages do not contain their contents directly. Instead, an intermediate `tar.gz` archive is created that contains actual contents of the package. This archive is then put into the `.deb` file directly. This allows full control over the process when the files from the intermediate package are unpacked and removed. The intermediate archive is unpacked from the `.deb` file by the package manager directly, while the actual files are unpacked by the maintainer scripts.

One downside of this approach is that application and release files had to be manually removed using maintainer scripts. Another one is that when an application is removed from a release, the package containing that application has to be manually removed. This should not be done from the maintainer scripts as it is probably a bad idea to invoke package manager from scripts invoked by the package manager.

4. Software update platform

Software Update Platform is responsible for performing updates on multiple devices and monitoring installed applications. Developers prepare *.deb* packages with applications and add them to a repository. When connection with device is established and information about planned update is written in database, platform will perform software update on that device.

During system design the following assumptions were made:

- Slow and unreliable connections – while developing we thought mainly about devices that are connected via GSM. This connection can be slow and may be interrupted easily.
- Possibility of NAT – geo-distributed mobile devices are typically not in a local network and can be located behind a NAT. Therefore the devices should connect to central server with public IP.
- Need for scalability – a distributed systems of considered class can consist of large number of devices. Our software management platform should be able to manage thousands of devices.
- Managing groups of devices, batch updates – user can create groups of devices so that our system can be used to manage more than one distributed system at once. User can send requests to the defined groups or to particular device.
- Simple, intuitive interface – the graphical management interface should be easy to use.

4.1. Architecture

General architecture of the Software Update Platform is presented in Figure 1.

The platform consists of two main parts:

- **Client application installed on mobile device.** It is assumed that client application is running on Erlang VM installed on some Linux distribution and *apt* is available. Client application connects with server and performs particular operations.
- **Management server.** The server consists of 3 components:
  - HTTP server
  - database
  - backend

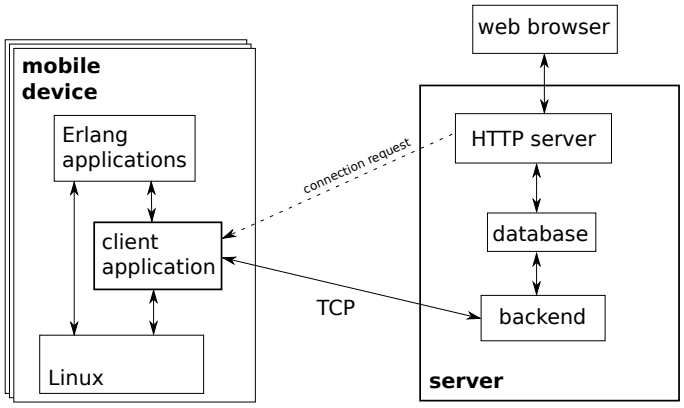

Figure 1. General architecture of the Software Update Platform.

HTTP server and user interface

Mochiweb, an Erlang-based lightweight Web server, is the server of choice. It is responsible for two main tasks. It manages user interaction through Web interface (serves content, performs user requests) and it is also a repository for *.deb* packages.

Web interface provides simple way to manage connected devices. A user can check last known state of device: IP, version of release, installed applications. There is also a convenient mechanism of managing groups of devices. Each device can be assigned to one or more category. Each scheduled job can concern single device or one of defined categories. Figure 2 shows exemplary view of device state.

The interface also provides easy mechanism of managing packages present in the repository. Previously prepared packages can be uploaded directly through the Web site.

Database

Mnesia, an Erlang native database stores information about:

- device and applications installed on it,
- jobs (e.g. update) for devices.

Figure 2. Device view from the user interface of the Platform.

Database is a connector between the Web interface and the backend.

Backend

The backend is the core of platform. It is Erlang application responsible for whole automatic device management. Every new device in platform has to connect with the backend. Information about all managed devices is stored in the database. The backend can monitor state of devices and perform operations on it.

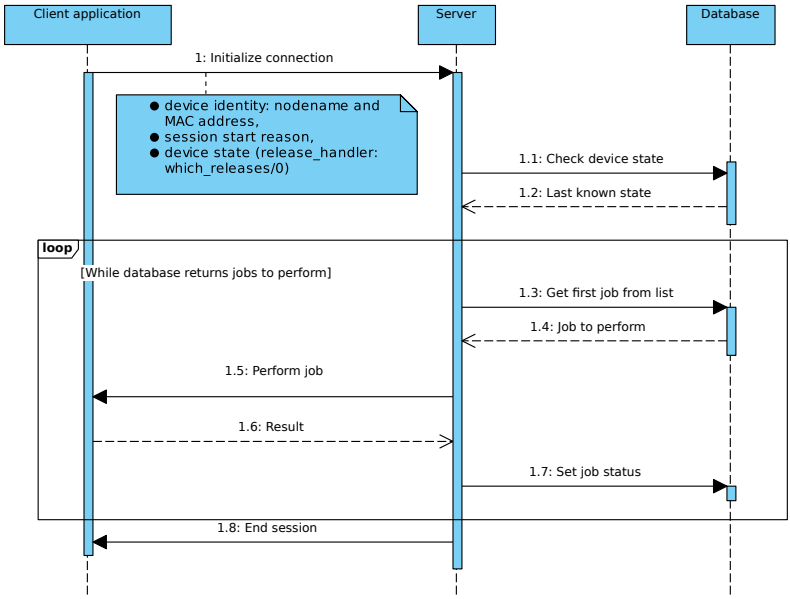

Figure 3. The diagram of communication between components of the system.

4.1.1. Device-server session

Sequence of operations performed during cooperation between a device and the server is presented in Figure 3. Firstly device initializes connection which carries following data:

- device identity: nodename and MAC address,
- session start reason,
- device states.

These pieces of information are used to unambiguously identify connected device. After connection is established, the server checks device state in the database. If there are any enqueued jobs, they are sent to device. The device returns result of performed jobs. This status is written to the database.

4.1.2. On-device upgrade logic

Updating software consists of two jobs:

- upgrade,
- check release.

Upgrade immediately returns with status `ok`. It stops periodic connection requests, closes session and uses *apt* to perform upgrade. It waits for *apt* to finish and checks its result. Then it restores periodic connection requests.

Check release reads value returned from *apt* and notifies server about changes in next session.

4.2. Development MModel for application

Software Update Platform proposes a model and provides some tools for development of Erlang software installed on the target systems. The main assumption is that the developer uses a tool called `rebar` for managing Erlang software (see [10] for more information). `rebar` is a script providing functionality for Erlang similar to what `maven` provides for Java development. This includes automatic generation of stubs for Erlang applications, build, `.appup` file generation, creation of complete, self-contained Erlang nodes (with a runtime), documentation etc. Generated Erlang nodes are used as a basis for `.deb` packages generation. This is implemented by a set of scripts provided by the Software Update Platform itself.

5. Conclusions and further work

Created system for dynamic updates of Erlang software on mobile devices provides unique functionality, which can simplify management of Erlang-based distributed systems. Software on remote devices can be upgraded without stopping updated applications. Amount of downloaded data is optimized, due to splitting a release into several packages related through dependencies. The system uses operating system package manager to install and upgrade Erlang applications, which is easier for people who had never used Erlang before.

There are several directions of further development of the platform. The most interesting are:

Support for other package managers. One obvious improvement for the platform would be support for more package managers, as Debian package format `.deb` was chosen arbitrarily for experimental development. Since its integration was successful, support for `rpm`-based package managers (like `yum`), `pacman` or `port` is planned in the future.

Communication security. Right now, communication between devices and the server is not encrypted. Wrapping device-server sessions in TLS and adding some authentication is another important potential feature for the platform.

Development towards more general management platform. The protocol used in communication between devices and the server is very general and extensible. New types of jobs can be easily added. Thus, the platform can be turned into more general management platform, providing a lot more of different functionality, like for example monitoring device status, configuring applications or viewing logs.

Acknowledgements

The research leading to this results has received founding from the Polish National Science Centre under the grant no. UMO-2011/01/D/ST6/06146.

References

[1] Reijers N., Langendoen K.: Efficient code distribution in wireless sensor networks. *Proc. of the 2nd ACM international conference on Wireless sensor networks and applications*, San Diego, CA, USA, pp. 60–67, 2003.

[2] Levis P., Patel N., Culler D., Shenker S.: Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. *Proc. of the 1st conference on Symposium on Networked Systems Design and Implementation*, vol. 1. San Francisco, California, 2004.

[3] Deng J., Richard H., Mishra S.: Secure code distribution in dynamically programmable wireless sensor networks. *Proc. of the 5th international conference on Information processing in sensor networks*, Nashville, Tennessee, USA, pp. 292–300, 2006.

[4] Han C., Kumar R., Shea R., Srivastava M.: Sensor network software update management: a survey. *International Journal of Network Management*, vol. 15(8), pp. 283–294, 2005.

[5] *Nokia Device management.* `http://europe.nokia.com/find-products/nokia-for-business/device-management`, 02.2012

[6] *Motorola Mobility Services Platform.* `http://www.symbol.com/category.php?category=159`, 02.2012

[7] Armstrong J.: *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[8] Logan M., Merritt E., Merritt R.: *Erlang and OTP in Action.* Manning Publications, 2010

[9] *Debian Policy Manual – Package maintainer scripts and installation procedure* `http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html`, 02.2012

[10] *Rebar: Erlang Build Tool* `https://github.com/basho/rebar/wiki`, 02.2012

Affiliations

Małgorzata Wielgus
AGH University of Science and Technology, Krakow, Poland, `malgorza@student.agh.edu.pl`

Przemysław Dąbek
AGH University of Science and Technology, Krakow, Poland, `przemyslaw.dabek@gmail.com`

Roman Janusz
AGH University of Science and Technology, Krakow, Poland, `roman@student.agh.edu.pl`

Tomasz Kowal
Erlang Solutions, Krakow, Poland, `tomekowal@gmail.com`

Wojciech Turek
AGH University of Science and Technology, Krakow, Poland, `wojciech.turek@agh.edu.pl`

Received: 21.02.2012
Revised: 6.07.2012
Accepted: 3.12.2012